The Kermit Project | *Columbia University*
612 West 115th Street, New York NY 10025 USA • kermit@columbia.edu
...since 1981

FTP Script Writing - How Do I Automate an FTP Session?

Most recent update: Fri Sep 10 17:52:20 2010

UNIX | WINDOWS | FTP CLIENT DOCUMENTATION | KERMIT SCRIPT LIBRARY

CLICK HERE to read about new FTP features in C-Kermit 8.0.206.

FTP SCRIPT : FTP AUTOMATION : AUTOMATE FTP : BATCH FTP : PROGRAMMABLE FTP : UNATTENDED FTP

> *This page is written for users of Unix operating systems -- Linux, Mac OS X, FreeBSD, AIX, HP-UX, IRIX, Solaris, etc. The Kermit FTP client is also available in Kermit 95 2.0 for Windows 9x/ME/NT/2000/XP, for which some of the applications, examples, and terminology used here might need minor adjustments (e.g. directory path syntax).*
>
> ***Also see:** Accessing IBM Information Exchange with Kermit for a discussion of making securely authenticated and encrypted FTP connections.*

Hardly a day goes by without an FTP automation question appearing in the newsgroups. Until now, the stock answers (for Unix) have been as follows (the options for Windows are sparse indeed):

1. Pipe commands into FTP's standard input. This works great when it works, but doesn't allow for synchronization, error handling, decision making, and so on, and can get into an awful mess when something goes wrong. For example, some FTP clients do special tricks with the password that tend to thwart piping of standard input or "here" documents into them. Also, the exact syntax of your procedure depends on which shell (sh, ksh, csh, bash, etc) you are using. Also, your password can be visible to other people through "ps" or "w" listings.

2. Put the commands to be executed into the `.netrc` file in your login directory in the form of a macro definition. Except for avoiding shell syntax differences, this is not much different than the first option, since FTP commands don't have any capability for error detection, decision making, conditional execution, etc. Note that the `.netrc` file can also be used to store host access information (your username and password on each host). It's a glaring security risk to have this well-known file on your disk; anybody who gains access to your `.netrc` also gains access to all the hosts listed in it.

3. Use Expect to feed commands to the FTP prompt. This improves the situation with synchronization, but:

   - It's cumbersome and error-prone, since it relies on the specific messages and prompts of each FTP client and server, which vary, rather than the FTP protocol itself, which is well-defined. Expect scripts break whenever the client or server prompts or text messages change, or if the messages come out in different languages.
   - You're still stuck with same dumb old FTP client and its limited range of function.

4. Use FTP libraries available for Perl, Tcl, C, etc. This might give direct programmatic access to the FTP protocol, but still offers limited functionality unless you program it yourself at a relatively low and detailed level.

Now there's a new alternative. The latest generation of **Kermit** software:

- C-Kermit 8.0 (or later)
- Kermit 95 2.0 (or later)

These programs include their own built-in FTP client, allowing FTP sessions to be automated using the same cross-platform scripting language we've been using for serial-port, modem, Telnet, and X.25 connections since the 1980s, in its advanced modern form:

`http://www.columbia.edu/kermit/ck80specs.html#scripts`

and has loads of features that you won't find in the regular UNIX FTP client:

`http://www.columbia.edu/kermit/ftpclient.html`

Here's a brief tutorial on writing C-Kermit FTP scripts. But the commands presented below are not just for scripts. You can also use them interactively, just as you would give commands to the regular UNIX or Windows FTP client, except that with Kermit you also get built-in help, context-sensitive help (if you type "?"), command recall, keyword and filename menus and completion, keyword abbreviation, and command shortcuts and macros.

Also see:

- Kermit Scripting Tutorial and Library
- Kermit FTP Client FAQ

EXAMPLE 1: SIMPLE ANONYMOUS DOWNLOAD

Let's begin with a simple anonymous connection to the FTP server at `xyzcorp.com` to download a file from the public `drivers` directory. The following table shows how to do this with a regular FTP client and with Kermit; the parts you type are underlined; the other parts are prompts from the shell or application:

Traditional FTP Client	Kermit FTP Client
$ ftp	$ kermit
ftp> open ftp.xyzcorp.com	C-Kermit> ftp open ftp.xyzcorp.com
Name: anonymous	Name: anonymous
Password: *user@somehost.com*	Password: *user@somehost.com*
ftp> cd drivers	C-Kermit> cd drivers
ftp> binary	C-Kermit> binary
ftp> get newdrivers.zip	C-Kermit> get newdrivers.zip
ftp> bye	C-Kermit> bye

As you can see, the procedures are practically identical. The main difference is that Kermit, since it can make many kinds of connections, must be told which kind to make ("ftp open"), whereas since FTP makes only one kind, it simply opens the connection the only way it knows how. Note, however, that any error handling in the procedures above is done strictly by the user. If an error message appears, the user reads it and decides how to respond. Other differences include:

- Kermit makes the connection in passive mode by default.
- On multiple-file downloads, Kermit does not prompt you for each file.
- Kermit does not need to be given BINARY or ASCII commands; it switches for each file automatically.
- Kermit has many functional advantages, listed HERE.

To make Kermit execute these commands automatically, just put them into a file:

```
ftp open ftp.xyzcorp.com /anonymous
cd drivers
binary
get newdrivers.zip
bye
```

and then tell Kermit to execute the file, which can be done in any number of ways (use the TAKE command at the `C-Kermit>` prompt; give the filename as the first command-line argument; or execute the file directly, like a shell script, as explained below). The first command (FTP OPEN) includes an "`/anonymous`" switch, which tells Kermit to log you in anonymously, automatically supplying your e-mail address as the password, thus bypassing the prompts. Executing this file is just like the interactive procedure but without engagement of your brain for decision making in case of errors.

Note, by the way, that there are simpler ways to accomplish the same task (download a single file anonymously), e.g. by giving Kermit the URL of the file on its command line:

```
kermit ftp://ftp.xyzcorp.com/drivers/newdrivers.zip
```

Now let's write the same procedure as a Kermit script, in which we illustrate some of Kermit's capabilities for detecting and reacting to errors:

```
#!/usr/local/bin/kermit +
ftp open ftp.xyzcorp.com /anonymous
if fail exit 1 Connection failed
if not \v(ftp_loggedin) exit 1 Login failed
ftp cd drivers
if fail exit 1 ftp cd drivers: \v(ftp_message)
cd ~/download
if fail exit 1 cd ~/download: \v(errstring)
ftp get /binary newdrivers.zip
if fail exit 1 ftp get newdrivers.zip: \v(ftp_message)
ftp bye
exit
```

Here's a brief explanation, line by line:

`#!/usr/local/bin/kermit +`

This is the "kerbang" line. If you want to run the script "directly" (as you would a shell script), this must be the first line in the script. The Kerbang line specifies the script interpreter to be Kermit rather than the shell. Substitute the appropriate C-Kermit 8.0 path name on your computer, and be sure to give the script file execute permission. For details see:

http://www.columbia.edu/kermit/ckscripts.html

`ftp open ftp.xyzcorp.com /anonymous`

This command attempts to open a connection to the FTP server on the computer whose IP hostname is `ftp.xyzcorp.com`. If the connection is successful, the command attempts to log in as user "anonymous", using your username and hostname (in email-address format) as the password.

`if fail exit 1 Connection failed`
: This checks whether the connection was made. If not, it prints an error message and exits with a status of 1, indicating failure. Here's where methods 1 and 2 above are lacking: if the connection fails, they'll just go ahead and try to execute the rest of the commands anyway.

`if not \v(ftp_loggedin) exit 1 Login failed`
: At this point we know we have a connection. This command checks whether login was successful by querying the value of one of Kermit's built-in FTP status variables, `\v(ftp_loggedin)`. If login failed, the script prints "Login failed" and exits with a failure code of 1.

`ftp cd drivers`
: This command asks the FTP server to change its directory to "drivers". Note that all commands for the FTP server begin with the word "ftp". That's because Kermit also has similar commands to be executed on the local computer, and other similar commands to be executed by a Kermit (not FTP) server.

`if fail exit 1 ftp cd drivers: \v(ftp_message)`
: Here we check the results of the FTP CD command. If it failed, a message such as "ftp cd drivers: No such file or directory" is printed, indicating the failing command and the FTP server's failure reason, and the script exits with a failure code. If we did not check for failure here, then the later download would fail or (worse) we might download the wrong file. Kermit's built-in `\v(ftp_message)` variable contains the most recent message from the FTP server. Note that when an FTP connection is active, Kermit's EXIT command also sends a BYE command to the FTP server and closes the connection.

`lcd ~/download`
: This command tells Kermit to change its own local directory to the `download` subdirectory of your login directory. LCD means "Local CD".

`if fail exit 1 cd ~/download: \v(errstring)`
: Here we check the local CD command in case it failed, for example because the directory could not be found. In this case, the local error message is printed rather than the FTP server's message (in UNIX, `\v(errstring)` is the error message that corresponds to the current value of `errno`).

`ftp get /binary newdrivers.zip`
: Now that both client and server are in the desired directories, we ask the server to send us the `newdrivers.zip` file in binary mode.

`if fail exit 1 ftp get newdrivers.zip: \v(ftp_message)`
: We check for failure in the normal manner.

`ftp bye`
: The file was received successfully. We log out and disconnect from the server...

`exit 0`
: And exit successfully (status code 0) from the script. If you don't include an EXIT command in the script, it won't exit; instead, it will issue C-Kermit's interactive command prompt and wait for you to type a command. The "0" is optional; Kermit's default exit status code is 0.

Let's say our script has been saved in a file called `getnewdrivers`. How to execute it? There are at least three ways:

1. At the `C-Kermit>` prompt, type "take getnewdrivers" (assuming the script is in C-Kermit's current directory).
2. At the shell prompt, type "kermit getnewdrivers" (assuming the script is in the current directory, and "kermit" is C-Kermit 8.0, and it is in your PATH).
3. At the shell prompt, type "getnewdrivers" (assuming the getnewdrivers script file is stored in a directory that is in your UNIX PATH; if it's not, type the full pathname).

In cases 1 and 2, the Kerbang line is not needed. Method 3 requires the `getnewdrivers` file to be in your PATH and that the Kerbang line of the script indicates the pathname of the C-Kermit 8.0 executable, and that the script file has execute permission:

```
chmod +x getnewdrivers
```

EXAMPLE 2: LOGGING IN AS A REAL USER

Here we modify our script to log in as a real user:

```
ftp open ftp.xyzcorp.com /user:olga /password:bigsecret
```

The rest of the script is the same, except perhaps now a full pathname is needed in the FTP CD command.

But it's a notoriously bad idea to put passwords in scripts or any other files (if this is news to you, please take it on faith). So how can the script log in as a real user without knowing the password in advance? There are lots of ways. The first is simply to have the script prompt for the password when it runs:

```
ftp open ftp.xyzcorp.com /user:olga
```

Just omit the /PASSWORD switch from the FTP OPEN command and Kermit prompts you for the password at the time it's needed (if it is), and you can supply it from your keyboard (it won't echo).

Alternatively, you can have Kermit prompt you for the password in advance. This might be appropriate when it's a long-running script and the FTP step doesn't happen until much later:

```
undefine \%p
while not defined \%p {
    askq \%p Password:
}
....
ftp open ftp.xyzcorp.com /user:olga /password:\%p
if fail exit 1 Connection failed
undefine \%p ; Erase password from memory
```

Here you see some "programming": variables and loops. We ask the user to type in the password using Kermit's ASKQ command ("ask quietly", i.e. don't echo the response). Since a password is required, the WHILE loop makes Kermit keep asking until it gets one, at which time it is assigned to the variable `\%p`. Then when the FTP OPEN command is given, `\%p` is specified as the password. Since `\%p` is a variable, it is replaced by its definition, which is whatever the user typed. (Normally, everything in Kermit that starts with a backslash indicates some kind of replacement -- a variable, a function call, the numeric representation of a character, etc.)

As a security precaution, the second "`undefine \%p`" command erases the password from memory immediately after it is used, like the comment says (trailing comments in Kermit are set off from the command by a semicolon (`;`) or number-sign (`#`) surrounded by whitespace).

Now suppose you want your script to run unattended when nobody is there to type in the password. This is a classic problem. One solution is to start the script early, type the password, and then have the script wait until the the desired time to do its work, using Kermit's SLEEP command, e.g.:

```
sleep 6000     ; Sleep 6000 seconds
sleep 23:59:59 ; Sleep until just before midnight
```

But what if you want the script to run periodically as a cron job, in which case there isn't even a terminal at which to type in the password? Well, that's a tough one, and it's one of the reasons for the appearance of secure FTP servers and Kermit's features for taking advantage of them. But that's another story, covered elsewhere:

http://www.columbia.edu/kermit/security.html

For example, if your version of C-Kermit was built with SSL/TLS security, and the server also supports SSL/TLS security, it is negotiated automatically. Various special commands can be used, but the only one that's required is SET AUTHENTICATION TLS VERIFY-FILE *filename*, that tells Kermit where to find the certificate file to be used to authenticate the FTP server.

EXAMPLE 3: PASSING PARAMETERS FROM THE COMMAND LINE

Let's generalize our little script to accept a host and a filename from the command line.

```
#!/usr/local/bin/kermit +
if < \v(argc) 3 exit 1 Usage: \%0 host file
ftp open \%1 /anonymous
if fail exit 1 \%1: Connection failed
if not \v(ftp_loggedin) exit 1 Login failed
lcd ~/download
if fail exit 1 cd ~/download: \v(errstring)
ftp get /binary \%2:
if fail exit 1 ftp get \%2: \v(ftp_message)
ftp bye
exit
```

Here we have simply replaced the host and file names by *variables*, `\%1` and `\%2`, whose values are set automatically by Kermit from the command-line arguments. These are similar to the `$1` and `$2` Shell variables.

Let's call this version of the script `getfile`, since it's not just getting new drivers any more; you can use it to get any file from any host that accepts anonymous logins. `\v(argc)` is a built-in variable that says how many "arguments" there were on the command line, including the name of the script itself.

Assuming `getfile` is installed as a Kerbang script in your PATH, now you can give commands such as these at the shell prompt (or in a shell script):

```
getfile support.scsicorp.com drivers/scsidrivers.zip
getfile kermit.columbia.edu kermit/archives/ckermit.tar.gz
```

If you run `getfile` without supplying the parameters it needs (host name and file name), it prints a usage message and exits with a failure code.

The command line arguments are passed to the script as:

`\%0`	The name of the script
`\%1`	The first command-line argument
`\%2`	The second command-line argument

Of course you can have more than 2 command-line arguments.

EXAMPLE 4: SUPPLYING DEFAULT PARAMETERS

If you don't want your script to fail if you pass it insufficient parameters on the command line, you can have it supply defaults for each missing parameter. There are two ways to do this; silently supply hardwired defaults, or prompt for missing parameters. The first way:

```
if not defined \%1 define \%1 ftp.xyzcorp.com
if not defined \%2 define \%2 newdrivers.zip
```

The second way:

```
while not defined \%1 {
    ask \%1 { Host: }
}
while not defined \%2 {
    ask \%2 { File: }
}
```

Or a combination:

```
if not defined \%1 ask \%1 { Host [ftp.xyzcorp.com]: }
if not defined \%1 define \%1 ftp.xyzcorp.com
if not defined \%2 ask \%2 { File [newdrivers.zip]: }
if not defined \%2 define \%2 newdrivers.zip
```

EXAMPLE 5: TRANSACTION PROCESSING CASE STUDY

Now that you know the basics of Kermit FTP scripting, let's look at a real-life application. *Transaction processing* refers to the communication from one computer to another of reservations, orders, votes, or other actions that have real (e.g. financial or political) consequences, usually from a large number of client computers to a central computer. Each transaction should take place exactly once -- not zero times, not two or more times -- otherwise a customer could be double-billed, or might not receive merchandise that was ordered; or votes might go uncounted or be counted multiple times; or a needed rental auto might not be waiting at the airport -- or six of them might be waiting!

A simple form of transaction processing is done by moving files from one computer to another, for example insurance claims from a pharmacy or doctor's office (the client site) to an insurance clearinghouse (the central site). A "watcher" process at the central site waits for files to appear in a certain directory and then processes them. In this case we want to make sure that each file is transferred completely and correctly, and exactly once, and furthermore:

- That the central-site process does not begin to process a file before it has fully arrived; and:
- That one client can't overwrite another client's transactions.

In a Kermit protocol client/server setting, all of this is handled quite nicely by Kermit's "atomic file movement" features. Unfortunately, not all of these features are available in FTP protocol, most notably a way to tell the FTP server to move or rename each incoming file automatically after it has fully arrived. However, we can accomplish the same thing with a Kermit FTP client script.

In this scenario, each client site has its own login ID on the central site to prevent file collisions between different clients, and also to provide an authenticated association between the uploaded files and the clients themselves. Each client ID at the central site has two subdirectories, `working` and `ready`. Client files (orders, votes, reservations, insurance claims, whatever) are uploaded to the `working` directory and then moved to the `ready` directory when the upload is complete. The move is "atomic" -- when the file appears in the `ready` directory, it appears all at once, not bit by bit; thus it is truly ready for processing the instant it is visible. The central-site "watcher" process periodically looks for files to appear in each client's `ready` directory, and when one does appear, moves it again, this time to its own area, and processes it. Thus any files in the client's `ready` directory are *waiting* to be processed and should not be disturbed. It is the client's responsibility to ensure that each file is sent completely, and sent only once, and that it is not disturbed after it is sent. It is the central site's responsibility to move files out of the ready directory and process them.

Our script expects the name of the file to send as its first command-line argument. We begin our script by checking the argument:

```
#!/usr/local/bin/kermit +
if not defined \%1 exit 1 Usage: \%0 filename
.filename := \fcontents(\%1)
.nameonly := \fbasename(\m(filename))
if not exist \m(filename) exit 1 \m(filename): File not found
if not readable \m(filename) exit 1 \m(filename): File not readable
```

Script and macro formal parameters (`\%1`, `\%2`, ...) are evaluated recursively, so that if their definitions contain variables, these are evaluated too, as many levels deep as variables are found. Since Kermit variables and other replacement quantities start with backslash

(\) this introduces an unfortunate conflict with DOS/Windows pathnames. Assigning the *contents* of `\%1` variable to a macro ("filename") forces one-level deep, rather than recursive, evaluation, and this allows our script to work with DOS or Windows file specifications as well as Unix ones. (For C-Kermit 9.0, see this.)

`filename` is the local name of the file to be sent, which can include a path -- i.e. it doesn't necessarily have to be in Kermit's current directory. `nameonly` is the name of the same file, but without the path. We use this to refer to the file's name on the server. The `\fbasename()` function strips any directory path from the filename, in case one was given (since the path is also stripped when sending the file's name to the FTP server).

Now we make the connection in the usual way:

```
undefine \%p
while not defined \%p {
    askq \%p Password:
}
ftp open centralsite.com /user:clientid /password:\%p
if fail exit 1 Connection failed
if not \v(ftp_loggedin) exit 1 Login failed
undefine \%p
ftp cd working
if fail exit 1 ftp cd working: \v(ftp_message)
lcd ~/upload
if fail exit 1 lcd ~/upload: \v(errstring)
```

Now we upload the file:

```
ftp put /delete \m(filename)
if fail exit 1 ftp put \m(filename): \v(ftp_message)
```

Notice that failure leaves the partial file (if any) in the working directory, where the central-site watcher process does not look for it. Thus transient failures do no harm. The script can be run again later. The /DELETE switch on the PUT command removes the source file after, and only if, it was uploaded successfully; this prevents it from being uploaded again (you could also have it moved or renamed). This way, even if the script is run again for the same file, it will fail immediately because the file is no longer there. Or, if a file of the same name is in the same place, it is a new file that *should* be uploaded.

Now we can move the uploaded file from the server's working directory to its ready directory (the syntax assumes a UNIX-like file system on server):

```
ftp rename \m(nameonly) ../ready/\m(nameonly)
if fail exit 1 ftp rename \m(nameonly): \v(ftp_message)
```

But wait, what if the destination file already exists in the server's `ready` directory? This would indicate that a previous transaction with the same name had not yet been processed. We should allow for this possibility:

`FTP CHECK` *filename*
: Succeeds if the file exists, fails if the file doesn't exist.

Here is the final version of our script:

```
#!/usr/local/bin/kermit +

; Verify command-line parameter (name of file to send)
;
if not defined \%1 exit 1 Usage: \%0 filename
.filename := \fcontents(\%1)
.nameonly := \fbasename(\m(filename))

if not exist \m(filename) exit 1 \m(filename): File not found
if not readable \m(filename) exit 1 \m(filename): File not readable

; Prompt for server password (OR USE SECURE FTP IF AVAILABLE!)
;
undefine \%p
while not defined \%p {
    askq \%p Password:
}
; Open the connection and log in
;
ftp open centralsite.com /user:clientid /password:\%p
if fail exit 1 Connection failed
if not \v(ftp_loggedin) exit 1 Login failed
undefine \%p

; Check if file of same name already exists on the server
;
ftp cd ready
if fail exit 1 ftp cd ready: \v(ftp_message)
lcd ~/upload
```

```
if fail exit 1 lcd ~/upload: \v(errstring)
ftp check \m(nameonly)
if success exit 1 \m(nameonly): Already exists in server ready directory.

; OK to send - cd to server's working directory.
;
ftp cdup
if fail exit 1 ftp cdup: \v(ftp_message)
ftp cd working
if fail exit 1 ftp cd working: \v(ftp_message)

; Now we upload the file and delete the local copy if successful.
;
ftp put /delete \m(filename)
if fail exit 1 ftp put \m(filename): \v(ftp_message)

; Move the uploaded copy to the ready directory
;
ftp rename \m(nameonly) ../ready/\m(nameonly)
if fail exit 1 ftp rename \m(nameonly): \v(ftp_message)

bye
exit 0
```

Call the script file `upload`, make sure the Kerbang line indicates the C-Kermit 8.0 path, give it execute permission, and then run it from the shell prompt as:

```
$ upload claim01.dat
```

If it didn't succeed, the error message will tell you why and you can take corrective action and run it again. If you run it again without taking corrective action, no harm is done -- either it will work or it will fail. If it works and you run it again on the same file, it will fail harmlessly because the original file is gone.

EXAMPLE 6: TRANSACTION PROCESSING - MULTIPLE FILES

The previous example showed how to upload a single file in a transaction processing environment. Let's generalize this to allow sending multiple files, assuming the same Working/Ready directory layout.

C-Kermit 8.0 also includes GET and PUT options (switches) to rename server files after successful transfer, whose use could shorten our transaction processing script, and are especially useful when transferring multiple files in a single operation: [M]GET or [M]PUT /SERVER-RENAME:*template*. Here is the previous script modified to accept a wildcard or directory name as `\%1`:

```
#!/usr/local/bin/kermit +

; Verify command-line parameter - source file(s) or directory
;
if not defined \%1 exit 1 Usage: \%0 filespec or directory name
if defined \%2 {
    echo "Fatal - Multiple arguments not supported."
    echo " You may give a single argument that is a filename,"
    echo " or a wildcard to match multiple files, or the name"
    echo " of a directory.  If you give a directory name, all"
    echo " files will be sent from that directory."
    exit 1
}
.filespec := \fcontents(\%1)
if directory \m(filespec) {
    lcd \m(filespec)
    if fail exit 1 - LCD \m(filespec) failed
    .filespec := *
}
if not \ffiles(\m(filespec)) exit 1 \m(filespec): No files match

; Prompt for server password (OR USE SECURE FTP IF AVAILABLE!)
;
undefine \%p
while not defined \%p {
    askq \%p Password:
}
; Open the connection and log in
;
ftp open centralsite.com /user:clientid /password:\%p
if fail exit 1 Connection failed
if not \v(ftp_loggedin) exit 1 Login failed
undefine \%p

; Make sure Ready directory is empty
;
```

```
ftp check ready/*
if success exit 1 Ready directory is not empty

; OK to send - cd to server's working directory.
;
ftp cd working
if fail exit 1 ftp cd working: \v(ftp_message)

; Now we upload the files, deleting each local copy and moving each
; uploaded copy when successful.
;
ftp mput /delete /server-rename:../ready/\v(filename) \m(filespec)
if fail exit 1 ftp mput \m(filespec): \v(ftp_message)

bye
exit 0
```

As written, the script accepts a single argument, which can be a filename, a wildcard to denote a group of files (which will need to be quoted if you invoke this script from the shell), or a directory name (in which case the the script will CD to the directory and the upload all the files from it). Of course the script could be modified to accept a list of arguments and/or various options.

All the work is done by the FTP MPUT command. The /DELETE switch says to delete each file that is sent successfully, and the /SERVER-RENAME: switch says to rename the file into the `../ready` directory as soon as it is fully received. `\v(filename)` is a built-in variable for use in file-group transfers that contains the name of the current file at the time it is being processed; it for use with switches such as /SERVER-RENAME that rename each file in a group on the fly.

In this case, we require that the `../ready` directory be empty, since FTP MPUT does not have a way to avoid renaming collisions a per-file basis. If there is any interest in such a feature, it can be added in a future release. In the meantime, per-file checking can be accomplished with a loop.

EXAMPLE 7: AUTOMATING SECURE FTP SESSIONS

Often when making secure connections, you are prompted interactively for certain information or permission to proceed. These prompts can stop an automated procedure. To avoid them, you must give the appropriate commands to disable them, and/or supply the prompted-for information beforehand. Here are a few hints:

- Make sure that SET TAKE ERROR and SET MACRO ERROR are both OFF. This is the default, but in case you have set either one of these ON in your script or initialization file, this makes the script halt on any kind of error. Normally you would want to check each operation for success or failure and take appropriate action.

- On SSL and TLS connections, you may be asked whether it is OK to proceed with a connection to server that presents a self-signed certificate. You can use the SET AUTHENTICATION SSL (or TLS) VERIFY or SET AUTH SSL (or TLS) CERTS-OK commands to avoid this prompt by not requesting a certificate from the peer.

- *(More to be added...)*

WHERE TO GO FROM HERE

The next step is to explore what other features are available. Kermit's feature set is rich, far beyond what you'd find in the typical FTP client. Suppose, for example, you want to **upload all the files that are less than five days old**, which might be any mixture of text and binary files, from a certain directory. Once the connection is made and desired directories are selected on the client and server, the command is surprisingly simple:

```
ftp put /after:-5days *
```

Notice there is nothing about text or binary mode in the command. That's because Kermit automatically switches into the appropriate mode for each file that it sends.

Or suppose you want to send **all the files that are larger than one million bytes** and whose names start with 'c' or 'w' except if the file's name is `core` or its name ends with `.log`:

```
ftp put /except:{{core}{*.log}} /larger:1000000 [cw]*
```

Or suppose you want to send all the files in an **entire directory tree**, which can include any combination of text and binary files, and have the same directory tree replicated on the FTP server, even if it is on a different operating system:

```
ftp put /recursive *
```

Now suppose that later, you want to refresh the same directory tree by uploading only those **files that changed since last time:**

```
ftp put /recursive /update *
```

Suppose you want to send a text file written in (say) German to another computer that uses a **different character set:**

```
ftp put /local-character-set:cp437 /server-character-set:latin1 Grüße.txt
```

Or suppose you want to continue uploading a very long file after a **previous upload attempt was interrupted in the middle:**

```
ftp put /recover verylong.tar.gz
```

Or suppose you want to synchronize a local directory from a remote one, even when you keep getting cut off, no matter how many tries it takes, without transferring any file that does not need to be updated, without transferring any file more than once, and without retransmitting any part of a file that was already partially received:

```
mkdir somelocaldirectory
cd somelocaldirectory

while true {
    ftp open foo.bar.com /user:myname /password:secret
    if fail exit 1 Can't reach host
    if not \v(ftp_loggedin) exit 1 FTP login failed
    ftp cd blah/blah/somepath
    if fail exit 1 Directory change failed
    while true {
 ftp get /recover /update *
 if success goto done
 if not \v(ftp_connected) break
    }
    ftp bye
}
:done
```

Or suppose you want to . . .

- Keep a record of your file transfers.
- Keep incoming files from overwriting existing files of the same name.
- Pass files through standard i/o filters as part of the transfer process.
- Upload a file with its permissions intact.
- Rename a group of files in a server directory.

All of this, and lots more, is easy to do with the Kermit FTP client, and it all can be automated.

CLICK HERE for more Kermit FTP script examples.

FURTHER INFORMATION

- The Kermit Project Home Page
- What Is Kermit?
- Kermit 95 for Windows
- C-Kermit for Unix
- Kermit FTP Client Overview
- Kermit FTP Client FAQ
- Kermit FTP Client Documentation
- The Kermit Script Library (FTP section)
- The C-Kermit User Manual
- C-Kermit 7.0 supplement to user manual
- C-Kermit 8.0 supplement to user manual

The Kermit Project / Columbia University / kermit@columbia.edu / 15 Oct 2003, updated 20 August 2011